February 5, 2025 Encoding

U+0000-U+007F		0xxxxxxx			
0080	07FF	110xxxxx	10xxxxxx		
0800	FFFF	1110xxxx	10xxxxxx	10xxxxxx	
10000	10FFF	11101xxx	10xxxxxx	10xxxxxx	10xxxxxx

Originally, in UTF-8, there were two ways to represent the same characters. This led to an exploit.

The left most column above, below the first row, could have four 0's prepended to them.

String comparison caused errors because of byte comparison `strcmp(str,"nobody")`

- The only files that nobody (special user, opposite of root) can read is the files the everybody can.

> Professor Eggert is a masochist talking about encoding 😭

Problems

- 11111000 → encoding error
  - Leading five 1's is invalid everywhere
  - this may bypass security filters by evading input validation in one layer but executing in another
- [unicode char.] [unicode char.] [10110111]
  - unexpected continuation byte
  - Attackers might inject standalone continuation bytes to corrupt data processing.
  - if a continuation byte appears **without a valid leading byte**, it is **invalid**.
- [unicode char.] [unicode char.] [11010101]
  - truncated sequence (either via EOF or via non-continuation bytes)
  - `110xxxxx` indicates the **start of a 2-byte sequence**
    - And it must be followed by 10xxxxxx
  - **Security Concerns**
    - Incomplete UTF-8 sequences could cause software to misinterpret input, potentially leading to **denial of service (DoS)** by crashing parsers.
    - Some systems may recover incorrectly, leading to **inconsistent interpretations** of the input.
- 11110000 1000000 1001000 10000 ???
  - overlong encoding
    - Confusing the defender with tricky encodings
  - **Exploit**: Attackers can use overlong sequences to **obfuscate malicious payloads**.
  - The **ASCII character** `A` **(U+0041)** should be encoded as `01000001` (1 byte).
  - But an **overlong encoding** would be:
    - `11000000 10000001` (invalid 2-byte form)
    - `11100000 10000000 10000001` (invalid 3-byte form)
- Unicode problems
  - homoglyph are chars that look the same, but are different
    - Latin letter o, Cyrllic letter o, and Greek letter o (omicron)
  - synoglyph are chars that are same, but look different (less of an issue than homoglyph from security perspective)
  - normalization: capital C + combination character turns into a C with a C with hook, Ç
    - normalization lists characters in a canonical order, if you normalize your character sequence, you must combo characters to get weird chars. in a specific order, else comparison bytes becomes tricky (like strokes in Chinese char.)
  - é is the combo of e + ' and stored as such

  Homos

  - µ (U+00B5) Micro Sign | μ (U+03BC) Greek Small Letter Mu (micro and mu are homos)
  - $\frac{1}{5}$ vs 1/5
  - German eszet
    - ß ⇒ SS
      - Unicode got lobbied into a bigger beta (typical Germans being betas)

Normalization is something only Unicode does, and there are 1-to-1 mappings.

Given a file : foo.txt

> How do we tell the encoding? A good guess is UTF-8 but this might be wrong
>
> Mac has metadata saying the specific encodings

RTL (right to left) vs LTR (left to right) ⇒ the bidirectional problem

- unicode sequence is in sequential order
  - it does not care if RTL or LTR
- display controls when things are RTL and LTR
- `\n` in a filename means ls shows something that looks like two files are there but in reality just one
- Unicode characters work in Python strings
  - String data types in python can be converted into a byte string with x.encode() where x is string
    - You can do the inverse with decode

```
x = #something
y = x.encode()
print(y)
z = y.decode()
print(z) # = print(x)
```

- you can read untrustworthy data in Python and try to decode() but you must be careful

Python the Sequel

- On the previous episode: we talked about Sequence types (creation of subsequences, etc.)

Mapping types (dictionary)

- keys can be any immutable value/object
  - The reason is this is fast is because the hashing function internally (hashing requires keys not to be mutable)
- dictionaries are implemented as hash tables

```
d = {}
len(d) # tells you number of items in dictionary
d.clear() # removes all items in dictionary
"""
VERSUS
"""
d = {} # creates a NEW dictionary and assigns it to d different than d.clear()
d.copy() # returns a copy f the o.g. dictionary but does NOT copy values or
         # keys
d.items() # returns all items as tuples
d.keys() # returns all keys
d.values() # return all values
d.update() # d = d Union d1 (set of keys unioned, values or d1 override d)
d.get(k[,v]) # get value in dict whose value is k safer than just direct access
d[k] # direct access, if k is not in dictionary (unlike get), get exception
del d[k] # delete item
```

- Dictionaries in pythons originally would have non-deterministic traversal because it was in the order of the hash table
  - Nowadays, it uses insertion orders
  - `d.popitem()`

Callable Types

```
f(a,b,c) # f in this situation must be a callable type

f = lambda a,b: a+b*b
def f(a,b):
   return a+b*b

g = f # take object f and assign it to g, now g is also callable
# you can stick callables into dictionaries and sequences if you want it
"""
Keyword arguments so you do not need to care about order of positional args.
"""
def f(a,*b): # b is 0 or more trailing arguments the caller will specify
             # b is a tuple containing the rest of the arguments after the first
```

```
def f(a,*b,**c): # * ⇒ tuple positional, ** ⇒ dictionary keyword args.
    # only one of * and ** args



"""
Classes are also callables.
a and b are parent classes of the subclass c
"""

class c(a,b):
    def m(self,other):
        return self.x + other

c() # → construct a new object of this class ⇒ constructor callable
```

- Class hierarchy is a directed acyclic graph (DAG)
  - DFS left to right for methods if not in immediate [sub]class

arctangent on midterm? kekw

Namespaces In Python

- A class is an object
- It has a member `__dict__`
  - that contains its names as a dictionary (under Python's control o.g. unless you redefine)
- By convenion __[] names a "private" to Python
- Some well defined private names,

```
class C:
    def __init__(self.[,args]):
        #[called on object construction]
    def __del__(self):
        #[called right before object cleanup]
        def __repr__(self):
        # full representation
        def __str__(self):
        #abbreviation used for debugging
```

```
def __cmp__(self,other):
    # comparison | return 0 or nonzero values
def __nonzero__(self):
    # defining if object can be considered nonzero matters
    # because p can be any object, by default it is nonzero
```

Python modules

- implemented by a source code file written in Python

```
if __name__ == '__main__'
# usually test suite for the module (if you run the module at the top level)
# if you do not run module at the top level, then you are importing the module
```

if you want to run `import foo` , we do the following:

1. create a new namespace
   a. that will eventually become foo
2. read from foo.py, execute that in new namespace (to prevent name collisions)
3. bind 'foo' to this namespace (in the invoker's namespace)

- Namespace is a runtime object and at runtime determine whether to import something

```
if x<0:
   import foo
```

Python packages

- A package hierarchy looks like a directory structure with a collection of modules
- Packages allow **hierarchical structuring** of modules, preventing name conflicts in larger projects.
- PYTHON PATH = /home/eggert/python:/home/millstein/py:/usr/local/cs/python3.11
    - list of directory names separated by colons

- Question: We have a package hierarchy that exists (a bunch of code arranged as a tree)
    - We also have a class hierarchy class c with parents a,b and child y
        - What distinguishes class vs package hierarchy trees?
            - TEST